Building a Customizable Embedded Operating System with Fine-Grained Joinpoints Using the AOX Programming Environment

Jiyong Park
School of Electrical Engineering and Computer Science,
Seoul National University, Seoul 151-744, Korea
+82-2-880-8370
parkjy@redwood.snu.ac.kr

Seongsoo Hong
School of Electrical Engineering and Computer Science,
Seoul National University, Seoul 151-744, Korea
+82-2-880-8357
sshong@redwood.snu.ac.kr

ABSTRACT
Aspect-oriented programming (AOP) has been successful in modularizing crosscutting concerns in complex software systems. In this paper, we present our aspect-oriented approach to building a highly customizable embedded operating system. This is a challenging task since embedded operating systems consist of intertwined concerns often implemented using a mixture of multiple programming languages including an assembly language. Furthermore, they often contain hand-optimized code that makes clear modularization extremely difficult. We provide a two-step approach that addresses these difficulties. First, we devised an aspect-oriented programming environment AOX (Aspect-Oriented eXtension). It supports both modularization and customization of complex software via a set of aspect-oriented mechanisms. AOX extends existing approaches in the sense that it is entirely programming language independent and provides fine-grained joinpoints. Second, using AOX, we built a customizable embedded operating system we call the HEART OS. It is highly configurable and very user-friendly. AOX has been implemented and integrated into the Eclipse IDE as a plug-in module. The HEART OS has also been implemented and ported to the XScale and x86 platforms. Our experience with AOX in building the HEART OS was very positive.

Categories and Subject Descriptors
D.2.6 [**Software Engineering**]: Programming Environments – Graphical environments

General Terms
Management, Design, Languages

Keywords
AOP, Operating Systems, Fine-granularity, Language Independence

1. INTRODUCTION
Embedded operating systems should adapt to a wide variety of hardware devices and application domains. However, various non-functional constraints inherent in embedded systems such as limited memory space, processing time and energy make one-fits-all operating systems impossible. Therefore, developers need to design embedded operating systems in a modularized and highly customizable fashion so that they can easily scale down their embedded operating system to a customized version that only contains exact features required.

The existing customizable operating systems often rely on conventional techniques such as macro preprocessing, conditional compilation, object-oriented programming or component-based development over micro-kernels. However, it is not always possible to achieve full customizability with these approaches since they fail to modularize crosscutting concerns that are inherent in most operating systems. Examples of such crosscutting concerns include various types of policies and path-specific optimizations, such as scheduling, memory management, synchronization and prefetching. It is thus unavoidable that code fragments for such concerns are scattered across the boundaries of source files, classes and components. Therefore, code fragments for such concerns necessarily cross the boundaries of source files, classes and components.

The most viable solution to this problem is the adoption of the aspect-oriented programming (AOP) technology [1]. In AOP, a crosscutting concern is modularized via a separate module called an aspect. In this paper, we present an aspect-oriented approach to building a highly customizable embedded operating system. Our contributions are two-fold. First, to render our approach reusable for a wide range of embedded software, we provide an aspect-oriented programming environment we call AOX (Aspect-Oriented eXtension). Instead of using existing AOP languages such as AspectJ and AspectC++, we have devised our own AOP language. It is designed to be independent of a base programming language and to have a fine-grained control over the weaving mechanism. Currently, AOX has been implemented and integrated into the Eclipse IDE as a plug-in. Second, to show the viability and effectiveness of AOX, we implement a customizable embedded operating system we call HEART OS using AOX.

The rest of this paper is organized as follows. In Section 2, we enumerate design requirements for our aspect-oriented programming environment. In Sections 3 and 4, we present AOX and HEART OS, respectively. Section 5 summarizes related work. Finally, we conclude this paper in Section 6.

2. DESIGN REQUIREMENTS AND OUR SOLUTION APPROACHES

Before delving into the details of AOX, we first enumerate its design requirements for supporting customizable embedded software development.

Language independence: An operating system commonly consists of various types of artifacts written in different programming languages. Core functionalities are usually written in C or C++, while machine-level functionalities, such as context switching, initialization and interrupt handling are written in assembly languages. In addition, makefiles are used to define software build processes, and linker scripts are employed to specify the memory layout of the target hardware. In order to modularize concerns that are scattered on such artifacts, we need to make our aspect model independent of languages used.

Fine-grained joinpoints: Due to performance constraints, operating systems often require highly optimized code. As a result, functions and data structures are deeply intertwined beyond module boundaries. For example, a function that creates a new process usually contains code fragments for seemingly unrelated features, such as context management, synchronization, signal, file and memory management. An interrupt handler array may contain elements for different devices. In order to enable such optimization, our aspect model needs to support fine-grained joinpoints.

Advice ordering and replacement: Programmers have to be able to decide the advising order of pieces of advice if a single pointcut is advised by multiple pieces of advice [2]. This is particularly important since the execution order of program statements, the placement order of fields in a structure and the order of elements in an array have important meanings in operating systems design. Along with the advice ordering, developers have to be able to replace existing pieces of advice with new ones. This is needed when developers want to design default pieces of advice that can be overridden by other pieces of advice. This effectively reduces the size of an operating system since replaced pieces of advice are removed.

Given these design requirements, our approach proposes the following solutions.

XML annotations: In our AOP language, an artifact is annotated by predefined XML tags. As the text that surrounds the tags is not parsed or interpreted, our model can be applied to artifacts written in any programming language. Also, this model achieves fine-grained joinpoints since the tag for a joinpoint can be declared anywhere in an artifact without restriction. We are well aware that our model may be unsafe since it does not prevent programmers from declaring joinpoints in inappropriate places or writing pieces of advice with illegal instructions. To overcome this drawback, our programming environment automatically performs weaving and compilation as a background task and immediately notifies programmers of errors caused by such unsafe joinpoints and pieces of advice.

Timestamp-based advice arrangement: In order to support the advice ordering and replacement, we devise the timestamp-based advice arrangement mechanism, in which ordering and replacement relationships among pieces of advice are represented in a two-level list structure. We store this structure in the most recently modified piece of advice to ensure that the ordering and replacement information is always up-to-date. To determine the most recently modified one, we associate each piece of advice with a timestamp value that is updated when its ordering or replacement relationship has been changed.

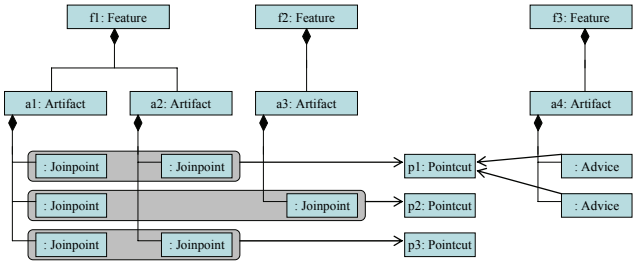

Figure 1. Object diagram shows relationships among joinpoints, pointcuts and pieces of advice.

3. AOX PROGRAMMING ENVIRONMENT

Our solution approaches in the previous section are incorporated in AOX. In this section, we give an overview of AOX by presenting the model of its AOP language. We also explain its weaving mechanism in detail. Finally, we present the implementation of the GUI and the weaver of AOX.

3.1 Model

In the AOX programming environment, a software system is modeled as a collection of *features*. A feature is an entity that encapsulates a specific concern of the system. It is implemented by one or more elements called *artifacts*. The artifacts are organized in a tree structure via an element called a *group*. When features are woven, a group and an artifact are mapped to a file and a directory in a file system, respectively.

A feature can modularize concerns that crosscut the file boundaries. This is done by the well-known advice-pointcut-joinpoint mechanism used in conventional AOP languages. In our model, an artifact consists of a text, *joinpoints* and pieces of *advice*. Here, a joinpoint represents a specific location (offset) in the text and a piece of advice represents a text fragment that can be copied to the joinpoints. A *pointcut* is defined as a named group of joinpoints. These three elements are associated as shown in Figure 1. In the figure, six joinpoints scattered in three artifacts and two features are captured by three pointcuts. Among them, one pointcut is advised by two pieces of advice defined in the third feature.

Features interact with each other only via *crosscutting interfaces*, each of which consists of related pointcuts. This idea is inspired by the idea of XPI [3]. A feature can implement a crosscutting interface by declaring joinpoints and associating them with the pointcuts in the interface. Similarly, the interface can be used by creating pieces of advice that advise some pointcuts in the interface.

The features, artifacts and interfaces are all represented in XML files stored in a hierarchy of directories in a file system that we call a *repository*. Specifically, inside an XML file for an artifact, a joinpoint is represented by a `<joinpoint>` tag. The location of the joinpoint is implicitly determined to be the point where the tag appears in the text. Similarly, `<advice></advice>` tag denotes a piece of advice. Any text other than the XML tags is considered source code and is not interpreted by AOX.

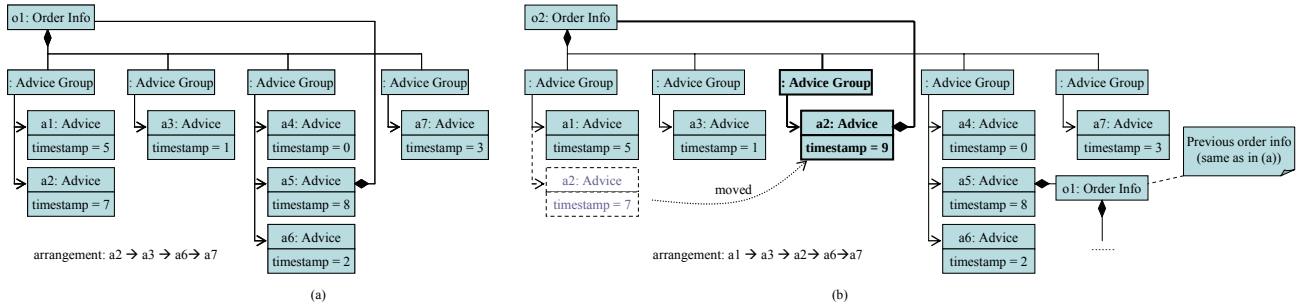

Figure 2. The timestamp-based advice arrangement mechanism. (a) The original arrangement. (b) *a2* is moved to the middle of *a3* and *a6*. The dotted lines and bold lines represent removed and added parts, respectively.

3.2 Weaving Mechanism

AOX uses text-based static weaving; the content (text fragment) of a piece of advice is copied into the locations where the associated joinpoints are declared.

When multiple pieces of advice are copied in a location, they must be copied by a predefined order. In AOX, this is done by the timestamp-based advice arrangement mechanism. In the mechanism, the order and replacement information is represented in a two-level list structure that we call *order info*. It is an ordered list of advice groups each of which is comprised of ordered lists of pieces of advice. Here, the first-level list determines the order of advice, while the second-level list determines the priority that controls the replacement of pieces of advice. In a second-level list, the last piece of advice in the list is assigned the highest priority and replaces the remaining pieces of advice. In Figure 2. (a), there are seven pieces of advice that advise the same pointcut and their arrangement is *a2*→a3→a6→a7. Pieces of advice *a1*, *a4* and *a5* are replaced, so they cannot participate in weaving. Using the two-level list structure, we can express any ordering and replacement information.

However, the problem of where to store the order info remains. When a piece of advice changes its location in the arrangement, new order info is created that reflects the new arrangement. In order to determine the most recent order info, we choose to store the new order info in the piece of advice that has been moved the most recently. To that end, each piece of advice has a timestamp field and it is updated when the piece of advice has been changed.

This process is also shown in Figure 4. (b). Here, *a2* has been moved between *a3* and *a6*. This new arrangement is described by new order info *o2*. The timestamp value of *a2* is set to 9 which is the highest timestamp value among the pieces of advice. Finally, *o2* is stored in *a2*. When weaving, the weaver search for the piece of advice that has the highest timestamp value and finds that it is *a2*. Then the weaver arranges the pieces of advice by using the order info stored in *a2*.

3.3 Graphical User Interface and Weaver

AOX is implemented as a plug-in module of the Eclipse IDE. It is designed to prevent programmers from having to deal with the complexities of XML files that implement the AOX programming language. Due to space limitations, we will not explain all aspects of the AOX plug-in in detail.

The GUI of the AOX programming environment mainly consists of AOX navigator, the text editor and the crosscutting viewer. The AOX navigator visualizes the structure of repositories. Programmers can navigate through features, artifacts, interfaces, configurations, pointcuts and so on. We also provide dedicated form-based editors for creating and modifying the above-mentioned elements, with the exception of the artifacts. For the artifacts, programmers can use any text editor that is provided by Eclipse or by other plug-in modules. AOX does not provide a special editor for artifacts. Instead, it augments the current text editor with graphical annotations and icons that represent joinpoints and pieces of advice.

Figure 3 shows a text editor augmented by AOX. Here, a vertical I-bar denotes a location where a joinpoint or a piece of advice is declared. They are distinguished via different colors. They can easily be added by dragging and dropping a pointcut from the AOX navigator to the specific location in the text editor. After that, a dialog is opened and the programmer is asked to choose whether a joinpoint or a piece of advice will be added in the location.

The details of joinpoints and pieces of advice are visualized and can be edited in the crosscutting viewer as shown in Figure 5. Here we see a list of joinpoints and pieces of advice that exist in the artifact that is currently being edited. In the Arguments/Content column, programmers can set argument values for a joinpoint or text content for a piece of advice.

The most important feature of the viewer is the ability to see and manipulate the arrangement of the pieces of advice. When a joinpoint or a piece of advice is expanded by the + button, all pieces of advice that advise the same pointcut are shown even if some pieces of advice come from other features. The pieces of advice are listed in the order that they will be woven together. In this way, programmers can see the result of weaving before the actual weaving is performed.

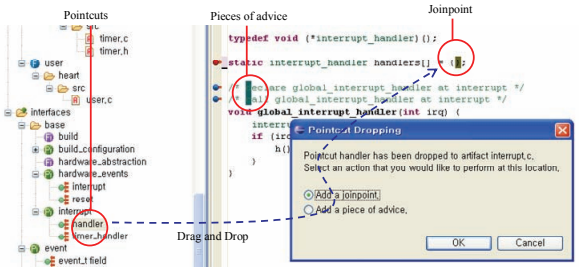

Figure 3. Screenshot of the text editor that shows graphical annotations for pieces of advice and joinpoints. The joinpoint of the *handler* pointcut is added to the text by drag and drop.

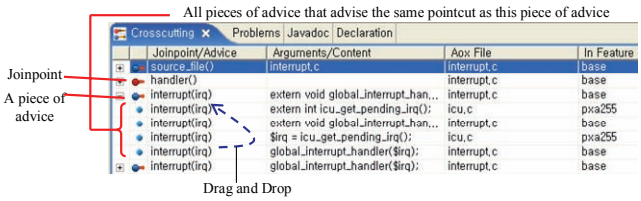

Figure 5. Screenshot of the crosscutting viewer.

The arrangement of pieces of advice can be changed by drag and drop. A piece of advice can be dropped between two adjacent pieces of advice or it may be dropped on top of another piece of advice to replace it. The replaced item is then shown as shadowed. Note that only the pieces of advice that are defined in the artifact that is currently being edited can be rearranged. This ensures that programmers cannot modify pieces of advice from other artifacts by mistake.

An artifact in a repository is woven into a file in an eclipse project. Weaving is activated by the *build* command that is invoked from the GUI. The command may be set to automatic so that the weaving is performed in the background whenever the repository is modified.

It is important that programmers be able to use project specific text editors, navigators, debuggers and useful functionalities, such as code assist and auto completion and the AOX plug-in simultaneously. This allows AOX to be applicable to any existing plug-in module and possibly to new plug-in modules that have not yet been developed.

4. HEART OS

Using AOX, we have built a customizable embedded operating system that we call the HEART OS (Highly Expandable Aspect-oriented Real-Time Operating System). Like most embedded operating systems, the HEART OS is designed as a library kernel and consists of heterogeneous files that are written in C, assembler, makefile and linker script.

The HEART OS consists of 15 features. When all features are enabled, the HEART OS provides functionalities similar to that of Nucleus or the uC/OS II real-time kernel. When only the minimal set of features is enabled, it becomes a very tiny operating system that only supports interrupt management. Currently, it is ported on the XScale and x86 platforms. To show the viability and effectiveness of AOX, we will now explain implementation details of two representative mechanisms of the HEART OS: scheduling and interrupt handling.

The scheduling mechanism is implemented as shown in Figure 6. Feature *arm*, which implements the behaviors specific to the ARM architecture, exposes pointcut *when_reset* at the reset handler function. Feature *scheduler* advises the pointcut to invoke *init_schedule()* during the reset.

The scheduling mechanism requires additional fields in thread control block (*struct thread*), e.g. priority and state. This is achieved by advising pointcut *new_field*. A joinpoint for the pointcut is located at the end of the declaration of *struct thread* in feature *thread*.

Those additional fields must be initialized when a thread is being created. Therefore, *thread* makes pointcut *when_created* available at the appropriate point in *thread_create()* function. By advising

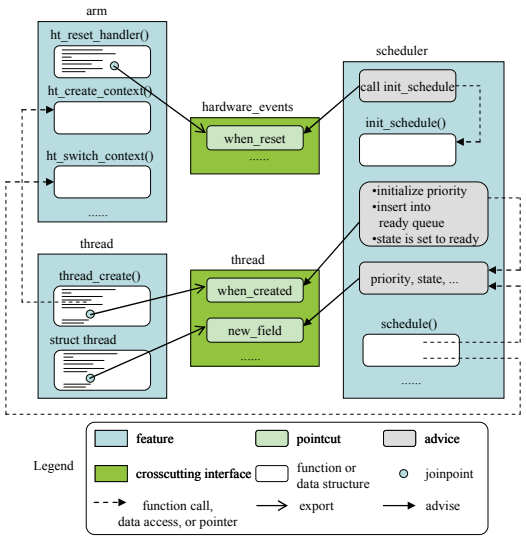

Figure 6. Scheduling mechanism in the HEART OS.

the pointcut, *scheduler* initializes the priority and state and inserts the thread into the ready queue. Note that the pointer to the thread structure is passed from *thread* to *scheduler* via the argument passing mechanism. Lastly, *scheduler* implements several scheduling functions as APIs, e.g. *schedule()* and *set_priority()*.

The interrupt handling mechanism is implemented as shown in Figure 7. First, *arm* makes a pointcut *when_interrupt* available at the point where hardware interrupt is first handled. The pointcut is advised by feature *base* so that *global_irq_handler()* function is invoked at every interrupt. Feature *base* is the default feature that is enabled in all configurations. It implements a generic OS architecture for the interrupt handler dispatching and provides a generic build script for the OS.

The function receives an IRQ number for the current interrupt and dispatches a handler from array *handlers[]* using the IRQ number as an index. In order for other features to add their own specific handlers to the array, the initialization part of the array is available through pointcut *handler,* which is defined in interface *interrupt*.

This pointcut is first advised by feature *pxa255*, which knows the number and the name of interrupt sources in XScale platform. However, since the feature may or may not provide handler routines for the interrupts, *pxa255* initially advises the *handler* pointcut with the pieces of advice that simply contain "`0`" (NULL). This means all interrupts are ignored by default. Among the interrupts, we have chosen to program *pxa255* to provide a handler for the timer interrupt (*timer0_handler()*), since it is the most important interrupt. In addition to that, as an example, we have programmed feature *user* to provide a handler for the serial device interrupt (*serial_handler()*). These two new interrupt handlers can be registered by advising pointcut *handler* and replacing the existing pieces of advice.

5. RELATED WORK

There have been many language extensions and mechanisms that support fine-grained joinpoints over existing AOP languages. These include the statement annotation [4], the test-based joinpoints [5] and Fluid AOP [6]. Among them, the statement annotation is closest to ours; a joinpoint can be declared by

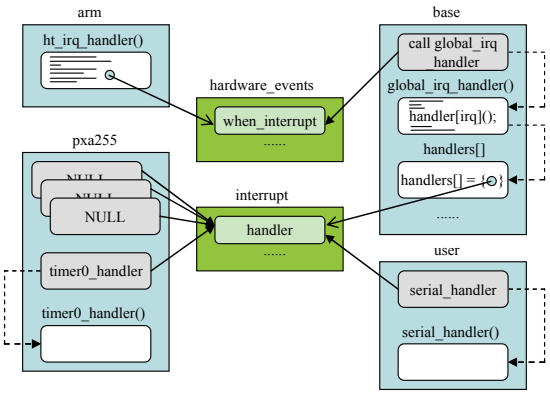

Figure 7. Interrupt handling mechanism in the HEART OS.

attaching an annotation to a statement in a method body. However, the mechanism relies on the language dependent annotation mechanism provided by Java, whereas we use language independent XML tags.

XVCL [7] is a language independent technique comparable to AOX; the base programming language is considered a simple text and the text is augmented using XML tags. Specifically, in XVCL, `<break>` and `<insert>` tags serve as a joinpoint and a piece of advice, respectively. However, it does not support the ordering of the `<insert>` tags. It also lacks the ability to modularize multiple concerns that crosscut each other, which is possible in AOX.

AspectJ and AspectC++ support aspect ordering in a different way from our approach. Using those languages, it is possible to declare partial orderings among aspects that will be collected to form a complete ordering. Nagy, et al. [2] proposed a mechanism that allow programmers to specify conditional and dynamic ordering among aspects. However, approach only support ordering at the level of aspects and cannot control the order among individual pieces of advice, nor do they support advice replacement.

There have been several GUI-based AOP tools that help programmers modularizing crosscutting concerns: Stellation [8], FEAT [9] and Bugdel [10]. They are similar to AOX in that programmers are not required to use the complicated pointcut designator of the conventional AOP languages. However, we have found that these approaches have a weak point as well; they do not work for an artifact that is augmented by other meta-language, such as a C source file decorated with the macro definitions. Also, they do not support selective enabling and disabling of concerns, and thus are difficult to use for developing customizable software systems.

In line with our HEART OS, there have been several research projects on using AOP technologies in modularizing operating system concerns. Coady, et al. [11] successfully modularized some path-specific optimizations concerns in the FreeBSD kernel using AspectC with low overhead. Lohmann, et al. [12] proposed CiAO which is an embedded operating system that modularizes its features using AspectC++. KLASY [13] is a dynamic aspect weaver which is used to add aspects dynamically to the NetBSD kernel.

6. CONCLUSION AND FUTURE WORK

In this paper, we have presented our aspect-oriented approach to building a customizable embedded operating system in two steps. First, we have devised the AOX programming environment with XML annotations and timestamp-based advice arrangement mechanisms. These mechanisms allow AOX to be independent of the base programming language and to enable fine-grained joinpoints. We have implemented AOX as a plug-in to Eclipse IDE. Although AOX is designed for our customizable operating system, we are confident that it can serve as a general programming environment for most embedded system software.

Using AOX, we have built a customizable embedded operating system we call the HEART OS. We have shown that our aspect-oriented approach is effective in modularizing crosscutting features in the operating system while satisfying the aforementioned requirements. Our experience with AOX in building the HEART OS was very positive.

We are now planning to enhance AOX. Specifically, it will be integrated with the variant management tools such as pure::variants so that AOX can be used for developing software product lines.

7. REFERENCES

[1] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming In *Proceedings of ECOOP*, 1997, 220-242.

[2] Nagy, I., Bergmans, L., and Aksit, M. Composing Aspects at Shared Join Points. In *Proceedings of NODe*, 2005.

[3] Griswold, W. G., Shonle, M., Sullivan, K., Song, Y., Tewari, N., Cai, Y., and Rajan, H. Modular Software Design with Crosscutting Interfaces. *IEEE Software* (2006), 51-60.

[4] Eaddy, M. and Aho, A. Statement Annotations for Fine-Grained Advising. In *Proceedings of RAM-SE*, 2006.

[5] Sakurai, K. and Masuhara, H. Test-based Pointcuts for Robust and Fine-Grained Join Point Specification. In *Proceedings of AOSD*, 2008.

[6] Hon, T. and Kiczales, G. Fluid AOP join point models. In *Proceedings of OOPSLA*, 2006.

[7] Jarzabek, S. Software Reuse Beyond Components with XVCL. In *Proceedings of GTTSE*, 2007.

[8] Chu-Carroll, M. C., Wright, J., and Ying, A. T. T. Visual separation of concerns through multidimensional program storage. In *Proceedings of AOSD*, 2003, 188-197.

[9] Robillard, M. P. and Murphy, G. C. Representing Concerns in Source Code. *ACM Transactions Software Engineering and Methodology,* 16, 1 (2007).

[10] Usui, Y. and Chiba, S. Bugdel: An Aspect-Oriented Debugging System. In *Proceedings of APSEC*, 2005.

[11] Coady, Y. and Kiczales, G. Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code. In *Proceedings of AOSD*, 2003, 50-59.

[12] Lohmann, D., Streicher, J., Spinczyk, O., and Schröder-Preikschat, W. Interrupt synchronization in the CiAO operating system: experiences from implementing low-level system policies by AOP. In *Proceedings of Workshop on Aspects, components, and patterns for infrastructure software*, 2007.

[13] Yanagisawa, Y., Kourai, K., Chiba, S., and Ishikawa, R. A dynamic aspect-oriented system for OS kernels. In *Proceedings of GPCE*, 2006.